REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-03-

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
07-01-2003	Final Technical Report	01/01/2002-3/31/2003

4. TITLE AND SUBTITLE
High Security Information System

5a. CONTRACT NUMBER
F49620-01-1-0329

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Lane, Brendan F.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of South Carolina
Department of Physics
Columbia, South Carolina 29208

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
USAF, AFRL
AF Office of Scientific Research
801 N. Randolph Street, Room 732
Arlington, VA 22203

NM

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR/PK3

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The overall focus of the University of South Carolina Critical Infrastructure Protection and Information Assurance Fellowship was in the area of comptuer security with a concentration in network traffic analysis. The Fellow addressed problems related to the developing approaches for fast and robust analysis of network traffic based on results obtained in complex system theory. To accomplish the research, it was important to develop and test wavelet technique for time-series analysis which could be naturally extended for the analysis of multidimensional time series. It also was desirable to find methods to speed up the existing time-consuming technique for obtaining characteristic time scales for the given chaotic time series. These requirements were the starting point for defining the set of mathematical problems to be solved.

15. SUBJECT TERMS
Network traffic analysis; complex system theory; wavelet technique; computer security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	none	20	Joseph E. Johnson
					19b. TELEPHONE NUMBER (Include area code) 803-777-8834

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

20031121 009

Air Force Office of Scientific Research
Grant Final Report

High Security Information System

Dr. Joseph Johnson, PI
Department of Physics
University of South Carolina

Dr. Brendan F. Lane, Fellow
University of South Carolina

Background of the Fellowship

Dr. Lane completed the requirements for a Ph.D. in Mathematics in the spring of 2001 and continued to work with the applied math group at the University of South Carolina until the end of calendar year. His dissertation and other research concentrated on image analysis with an emphasis in image registration. With a strong background in wavelet theory, numerical analysis, and computer programming, Dr. Lane was appointed the CIPIAF Fellowship in late 2001 and joined Dr. Johnson's research group in January 2002.

Brief Summary

This report will describe the work and research that has been accomplished by the University of South Carolina (USC) recipient of the Critical Infrastructure Protection and Information Assurance Fellowship (CIPIAF). The overall focus of the Fellowship was in the area of computer security with a concentration in investigating techniques and strategies of network traffic analysis. Providing network managers, computer scientists, and security administrators the ability to recognize network traffic anomalies efficiently would be extremely helpful and a potential starting point for real-time detection and possible avoidance of security breaches such as denial of service attacks and intrusions.

The fellow addressed problems related to the developing approaches for fast and robust analysis of network traffic based on the results obtained in the complex system theory. To do this it was important to develop and to test wavelet technique for time series analysis which could be naturally extended for the analysis of multidimensional time series. Also, it was desirable to investigate speeding up the existing time-consuming technique, based on the mutual information calculations, for obtaining characteristic time scales for the given chaotic time series. These requirements were the starting point for defining the set of mathematical problems to be solved.

The Fellow's work was composed of several different projects that investigated potential techniques which eventually could be incorporated into the afore-mentioned overall process of network traffic data analysis. The time involved with each of these projects included a period of investigative reading, writing computer code that mimicked previous

algorithms and results, and writing programs that implemented new techniques and produced data to justify and support its usage. Along with these projects, the year was also spent communicating with other researchers. Along with interacting with researchers in the USC community, attendance at three conferences provided an opportunity to meet people from around the nation who are investigating similar problems.

Research Projects

This section of the report will describe the different projects that were completed during the Fellowship.

Rossler and Lorentz attractors

The purpose of this project was to generate a set of data to be used in future applications. Using well-understood non-linear dynamic systems, the data was chosen to model some of the properties of network computer traffic. With properly selected parameters, the Rossler and Lorentz systems provide bounded three-dimensional pseudo periodic data streams. Our application will consider the solutions of these systems to be three states in which a particle's motion (changing states) is dependent.

$$\begin{aligned} x' &= \sigma(y - x) \\ y' &= rx - y - xz \\ z' &= xy - bz \end{aligned}$$

Figure 1: Lorenz system of equations.

$$\begin{aligned} x' &= -y - x \\ y' &= x + Ay \\ z' &= B + xy - Cz \end{aligned}$$

Figure 2: Rossler system of equations.

One avenue of research in network traffic analyzes either the TCP or IP packet information that accompanies the data that travels on the Internet. Fields from these packets are fixed length binary information and are thus bounded. In regards to using a dynamical system to generate the data stream is to subscribe to the assumption that the traffic flow is not completely random. The systems that were used are well behaved but still chaotic in nature and although the flows follow a general tendency, the flows are not periodic.

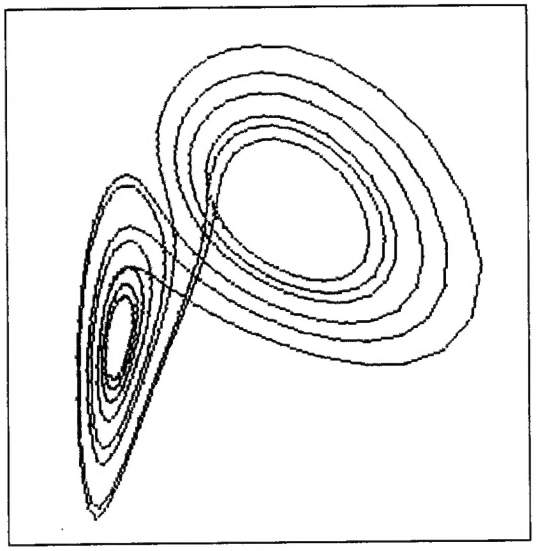

Figure 3: Attractor from Rossler system.

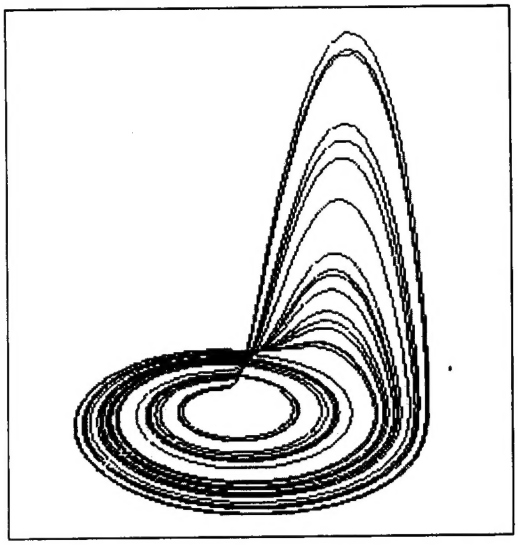

Figure 4: Attractor from the Lorenz system.

Picard Method

While generating the solutions for the Rossler and Lorenz systems, a high order method was implemented to avoid the lack of stability in the simpler step methods predicting a solution. Since these systems are first order ordinary differential equations, an approximation to the exact solution can be found by repetitive scheme of using the initial value (or subsequent solutions) and adding the product of the instantaneous velocity (which is provided in the statement of the equations) and a certain time step. To remain stable, low order approximations are forced to use smaller time steps, hence requiring more computations when approximating the path of a particle over a fixed amount of time.

The Picard method is one method of solving first-order differential equations that are in algebraic form. Being in algebraic form means that the system of equations is defined where the derivatives of each variable equal to a polynomial of the variables of the system. In this research, the Rossler and Lorenz systems are already in algebraic form (see equations in previous section). The Picard method can be implemented to approximate with any order, thus allowing for longer time steps and remaining stable. Any first-order system of ordinary differential equations can be modified into an algebraic form by introducing additional equations to eliminate the non-algebraic terms.

Along with implementing the Picard method for the two systems, a program was written to simplify the use of the Picard method in future work which allows for users to define their algebraic system of definitions easily and to specify the order of approximation. Without this symbolic language (similar to reverse Polish notion), one would have to carry out numerous computations just to set up a single problem for a specified order of approximation.

The following is the subroutine that defines the Rossler attractor. The variable A is a vector holding each of the polynomial derivative definitions while the variable V is a vector of the variable values. One should note that order of approximation is not part of the system definition, so no additional changes are required if a higher order is needed.

```
void Rossler()
{
   double a, b, c ;
   a = 10 ;
   b = 2 ;
   c = 4 ;

   PushPoly( &A[0], &V[1] ) ;
   PushPoly( &A[0], &V[2] ) ;
   PushOperand( &A[0], 'A' ) ;
   PushScalar(  &A[0], -1 ) ;
   PushOperand( &A[0], '*' );

   PushPoly( &A[1], &V[1] ) ;
   PushScalar( &A[1], a );
   PushOperand( &A[1], '*' );
   PushPoly( &A[1], &V[0] ) ;
   PushOperand( &A[1], 'A' );

   PushPoly( &A[2], &V[0] ) ;
   PushScalar( &A[2], -c ) ;
   PushOperand( &A[2], '+' );
   PushPoly( &A[2], &V[2] );
   PushOperand( &A[2], 'M' );
   PushScalar( &A[2], b );
   PushOperand( &A[2], '+' );
}
```

Entropy of systems

This project investigated methods and techniques of measuring data streams of information. In this project, the goal was to create a good method to measure the difference between one set of ordered values against the same set shifted (time delayed) by a fixed amount *k*. Traditionally, the autocorrelation function has been used to measure the independence between these two sets of values. Other research (P.S. Shaw and A.M. Frasier/H. L. Swinney) has stated that the mutual information can be used to better estimate the zero of the autocorrelation function.

The autocorrelation function is defined for a set of ordered values and a given offset value k as the following way.

$$\mathrm{AC}_k(\{y_i\}) = \frac{\sum(y_i - \overline{y})(y_{i+k} - \overline{y})}{\sum(y_i - \overline{y})^2}$$

Figure 5: Definition of the autocorrleation function

The autocorrelation function is scale and shift invariant and has the range from –1 to 1.

The first part of the project concentrated on system entropy and the mutual information function of a probability density function. If given a discrete set of events each have its own associated probabilities, the entropy of the system is equal to the average amount of information and is formulated by the following weighted average:

$$E(\{p_i\}) = \sum_i p_i \log \frac{1}{p_i}$$

Figure 6: Definition of Entropy

In terms of information, this weighted average is the balance between the likelihood of an event happening and the information one gains from it happening. On one hand, if there exists only one event with probability 1 of occurring, then there is no information that is gained when the event occurs. On the other hand, if there are 16 events with equal chance, then the entropy of this system is 4; consider that it would take four bits to encode the different events and on average you would gain these four bits if any one event occurred.

Another function in measuring a probability density function of two dimensions is with the mutual information function. The mutual information function uses not only the probability of the discrete events but also the marginal distributions of each variable and is formulated by the following:

$$M(\{p_{ij}\}) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{p_{i,\cdot} p_{\cdot,j}}$$

Figure 7: Defintion of Mutual Information

The mutual information function can be re-written using the entropy function and shows that the mutual information function is measuring the difference between the information of the entire system versus the sum of the information gained by looking at the marginal probabilities.

$$M(\{p_{ij}\}) = E(\{p_{i,\cdot}\}) + E(\{p_{\cdot,j}\}) - E(\{p_{ij}\})$$

Figure 8: Mutual information in terms of entropy

Relating this function back to probability theory, if the indices are treated as random variables, the mutual information function is zero when the random variables are independent.

In this project, the probability density functions were created from the data streams of the Rossler and Lorenz systems. Given an integer *k* which reflects the periodicity, a significantly large set of points *A* from one of the previously mentioned data set and considering one of the variables *w* (or any projection of the independent variables) of the data stream, the probability density function F_k is defined by the following:

$$F_k(x, y) = P((x, y) = (w_i, w_{i+k}) \text{ where } w_i \in \text{Proj}(A))$$

Figure 9: Probablity density function for measuring time delay dependence in data flow information.

In practice, the above definition is modified to measure the probability over a neighborhood of points rather the equality shown above. Using this practice, the function then reflects approximately the proportion of observations occurring in the neighborhood of *(x,y)*. To determine this function, the range of values was segmented into equal sized squares and with one pass through the data, an intermediate histogram reflecting the number of observations that landed in each square was constructed. With this histogram, the probabilities were computed by dividing each value of the histogram by the total number of observations. This two-dimensional probability function can be used to quickly compute the autocorrelation function and the mutual information function.

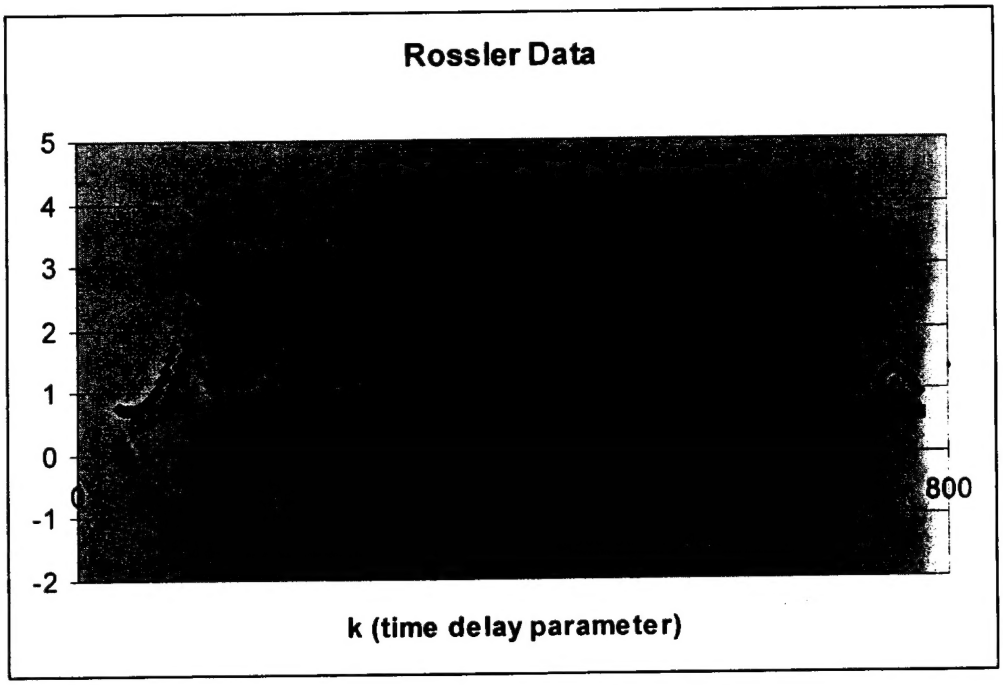

Figure 10: Comparison of the mutual information function and the autocorrelation function.

Using the Rossler data, a strong relationship can be seen between the graphs of the mutual information function and the autocorrelation function as the time delay parameter varies. The peaks in the mutual information occur at approximately the half-integer multiples of the estimated periodicity of the Rossler attractor. These locations also correspond to the local minimums and maximums of the autocorrelation function. Between the peaks, the local minimums of the mutual information function occur at the zeros of the autocorrelation function.

Besides showing the relationship, work to approximate the data (namely the histogram) using wavelets to reduce the number of computations was investigated. By uniformly increasing the size of the neighborhoods by a factor of 2 in the definition of the modified probability density function, the number of computations reduces by a factor of 4.

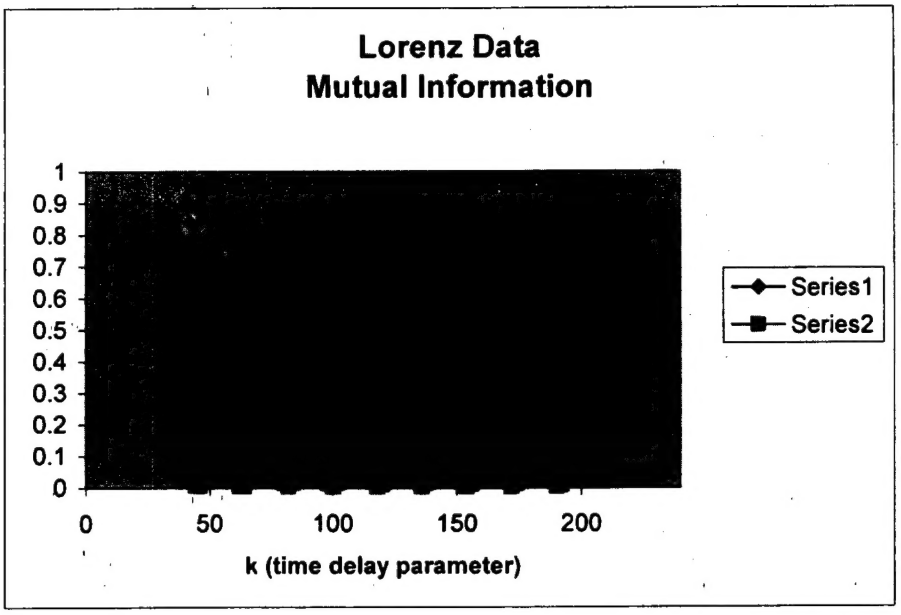

Figure 11: Comparison of different resolutions in computing the mutual information function using the Lorenz attractor data set.

Comparing the graphs of the mutual information function of different resolution shows that the locations of the minimum values occur at approximately the same k values. With this fact, wavelet coefficient thresholding was implemented to incorporate the use of coarser resolution in certain areas of the density function's range to reduce the number of computations. The use of wavelets will be beneficial in future work to compare more than two data flows at a given time. By first approximating a multivariate probability density function with wavelet thresholding, then using the tree structure of the coefficients in the computation phrase of the mutual information function should dramatically reduce the number of computations by avoiding calculations where the coefficient tree is sparse

This project is still ongoing and is work that should be explored in the future. The packet information from Internet data traffic is multidimensional and methods of analyzing more than two fields of information will be needed. Other methods that can show dependence or independence with less computational effort should be explored as well. Avenues that seem to show promise would be to relate the data to smoothness spaces (like Sobolev or Besov spaces). In working with smoothness spaces, there is a natural link to information theory in the area of complexity and relating complexity back to the smoothness classes.

New contacts and new research

During the fellowship period, an effort was made to develop new contacts with others in the research community and to become acquainted to some degree in new research areas. Within USC, new relationships were built with people in the Department of Physics, Department of Statistics, and Department of Electrical Engineering. Also there were some new contacts (for example, from University of Wisconsin, James Madison

University, and Naval Research Labs) made at conferences that were attended during the year or during visitations.

During the year, three conferences were attended. Each of the conferences was very useful in understanding the current research that is being done in the areas of computer security. First in March 2002, the fellow and Dr. Johnson attended the OASIS conference that focused on computer security and information integrity.

In November 2002, the IDR (Ideal Data Representation) Fall Conference was held at USC. The IDR group is a NSF-sponsored consortium of universities involved with developing data representations. Several talks were given related to the topic of network events and traffic anomalies. Different groups were working at problems similar to the ones that were study during the fellowship year. Other groups were interested in the behavior and anomalies of the volumes of network traffic rather than the individual packets.

Lastly, the fellow attended the Interface 2003 symposium in Salt Lake City March 2003. Sponsored by The Interface Foundation of North America, the symposium focused on the topics of security and infrastructure protection with subtopics "statistical analysis and probabilistic modeling of Internet traffic" and "statistical issues in computer security." Research in detecting denial of service attacks and modeling network traffic were of particular interest. The coverage of information at the last conference would have been extremely beneficial if it had been held at the beginning of the fellowship year, but the information gained at the conference did provide assurance that the problems that had been studied are of importance and that others are investigating and still trying to add to the current state of this research topic.

Conclusion

With respect to the goals of the fellowship, the past fifteen months have been very productive in terms of new research. The questions that the Fellow investigated are of importance and do not have simple answers Therefore, further investigations need to be done to develop reliable and efficient mathematical techniques for network traffic analysis. Although the fellowship period is over, hopefully there will be time to continue this research. Nevertheless, the research completed for this project could be considered as a good first approximation for the solution of the complicated problems of the network traffic analysis.

Appendix – Computer code

The following is the computer code that was written to assist in the research during this research project. This compilation of code is given to provide a general idea of how some of the data was generated and how the algorithms work. Some routines have not been included.

MutualInfo.cpp

Computes the mutual information function and autocorrelation function for a given data stream at different time delays.

```
int main(int argc, char **argv)
{
        int i ;
        int Variable = 2 ;
        int DataSize ;
        int bytes = 0;
        double xvalue, yvalue, minfo, hinfo, correl ;
        double T ;
        double TimeStep, OrbitLength ;

        glutCreateWindow("Data Analyst");

        DA = MakeDataAnalyst( 8, 8 );
        SetDataAnalyst( DA, -100, 100, -100, 100 );

        if( OUTPUT_FLAG ) ofp = fopen( OutFileName, "wb" ) ;

        TimeStep = 100.0/M_PI ;
        OrbitLength = 200.0 ;

        Orbit = MakeOrbit( ) ;

        for( T = 0.2 ; T < 2.1 ; T += 0.01 )
        {
                DataSize = int (OrbitLength * T );
                MakeOrbitBlock(Orbit, DataSize ) ;
                SecondBlock(Orbit, DataSize);

                OpenOrbitFile(Orbit, OrbitFileName) ;
                HeadOrbitFile(Orbit) ;
                getFirstBlock(Orbit) ;
                SaveBlock(Orbit); getNextBlock(Orbit);

                bytes = 2*Orbit->NoBytes ;
                while( !AtEOF(Orbit) && (bytes < TotalDataPoints ))
                {
                        bytes += Orbit->NoBytes ;
                        for( i = 0 ; i < Orbit->NoBytes ; i++ )
                        {
                                xvalue = OrbitData2(Orbit, i, Variable );
                                yvalue = OrbitData(Orbit, i, Variable );
                                DAStoreData( DA, xvalue, yvalue );
                        }
                        SaveBlock(Orbit); getNextBlock(Orbit);
                }
                getFirstBlock(Orbit) ;
                for( i = 0 ; i < Orbit->NoBytes ; i++ )
                {
                        xvalue = OrbitData2(Orbit, i, Variable );
                        yvalue = OrbitData(Orbit, i, Variable );
                        DAStoreData( DA, xvalue, yvalue );
                }

                CloseOrbitFile(Orbit) ;

                DANormalize( DA ) ;
                printf( "DataPts: %d  BadPts: %d \n", bytes, DA->BadData ) ;
                minfo = MutMatrix() ;
                correl = AutoCorrel( ) ;

                if( OUTPUT_FLAG )
                        fprintf( ofp, "%d\t%lg\t%lg\t%lg\n", DataSize, T, minfo, correl ) ;
```

```
                DestroyBlocks( Orbit );
                DAEraseData( DA ); bytes = 0 ;

        }
        DestroyOrbit( Orbit );
        if( OUTPUT_FLAG ) fclose( ofp ) ;

        return(1)  ;
}

double AutoCorrel()
{
        double minfo ;
        mat = (WVmatrix *) malloc( sizeof( WVmatrix ) );
        mat->size     = wvVaNewIndexVector(2, DA->height, DA->width );
        mat->bufSize  = wvVaNewIndexVector(2, DA->height, DA->width );
        mat->buf       = DA->Bins ;
        mat->ownBuf   = (wvBool) 0 ;

        minfo = AutoC( mat ) ;
        free( mat ) ;
        return( minfo ) ;
}

double AutoC( WVmatrix *mat )
{
        int i, j, height, width ;
        double log2 ;
        double value = 0.0 ;
        double IJvalue, IIvalue, Ivalue ;
        double cellvalue ;

        log2 = log(2) ;
        height = wvIndexVectorSub( wvMatrixSize( mat ), 0 ) ;
        width  = wvIndexVectorSub( wvMatrixSize( mat ), 1 ) ;

        Ivalue = 0.0 ;
        IIvalue = 0.0 ;
        IJvalue = 0.0 ;
        for( i = 0 ; i < height; i++ )
                for( j = 0 ; j < width ; j++ )
                {
                        cellvalue = mat->buf[i*width+j] ;
                        Ivalue  +=   i*cellvalue ;
                        IIvalue += i*i*cellvalue ;
                        IJvalue += i*j*cellvalue ;
                }

                value = (IJvalue - Ivalue*Ivalue) / (IIvalue - Ivalue*Ivalue) ;
                return( value ) ;
}

double MutMatrix()
{
        double minfo ;
        mat = (WVmatrix *) malloc( sizeof( WVmatrix ) );
        mat->size     = wvVaNewIndexVector(2, DA->height, DA->width );
        mat->bufSize  = wvVaNewIndexVector(2, DA->height, DA->width );
        mat->buf       = DA->Bins ;
        mat->ownBuf   = (wvBool) 0 ;

        minfo = MutualInfo( mat ) ;
        free( mat ) ;
        return( minfo ) ;
}

double MutualInfo( WVmatrix *mat )
{
        int i, j, height, width ;
        double log2 ;
        double value = 0.0 ;
```

```
        double Pvalue, PXvalue, PYvalue ;
        double cellvalue ;

        log2 = log(2) ;
        height = wvIndexVectorSub( wvMatrixSize( mat ), 0 ) ;
        width  = wvIndexVectorSub( wvMatrixSize( mat ), 1 ) ;

        Pvalue = 0.0 ;
        for( i = 0 ; i < height*width ; i++ )
        {
                cellvalue = mat->buf[i] ;
                if( cellvalue > 0.0 ) Pvalue += cellvalue*log( cellvalue )/log2;
        }

        PXvalue = 0.0 ;
        for( i = 0 ; i < height ; i++ )
        {
                value = 0.0 ;
                for( j = 0 ; j < width  ; j++ )
                {
                        cellvalue = mat->buf[i*width+j] ;
                        value += cellvalue ;
                }
                if( value > 0.0 ) PXvalue += value*log( value )/log2 ;
        }

        PYvalue = 0.0 ;
        for( j = 0 ; j < width  ; j++ )
        {
                value = 0.0 ;
                for( i = 0 ; i < height ; i++ )
                {
                        cellvalue = mat->buf[i*width+j] ;
                        value += cellvalue ;
                }
                if( value > 0.0 ) PYvalue += value*log( value )/log2 ;
        }

        value = Pvalue - PXvalue - PYvalue ;
        return( value ) ;
}

double H_function( WVmatrix *mat )
{
        int i, height, width ;
        double log2 ;
        double value = 0.0 ;
        double Pvalue ;
        double cellvalue ;

        log2 = log(2) ;
        height = wvIndexVectorSub( wvMatrixSize( mat ), 0 ) ;
        width  = wvIndexVectorSub( wvMatrixSize( mat ), 1 ) ;

        Pvalue = 0.0 ;
        for( i = 0 ; i < height*width ; i++ )
        {
                cellvalue = mat->buf[i] ;
                if( cellvalue > 0.0 ) Pvalue += cellvalue*log( cellvalue )/log2;
        }

        value =  - Pvalue  ;
        return( value ) ;
}
```

DataAnalyst.cpp

These routines handle the data type DataAnalyst. The DataAnalyst is essentially a two dimension grid that records the number of observations that are seen in a data set. When normalized, the DataAnalyst grid becomes a probability density function.

```
typedef struct _DA
{
        double *Bins ;
        int height, width ;
        double dx, dy ;
        double x0, x1, y0, y1 ;
        int BadData ;
} DataAnalyst ;

DataAnalyst *MakeDataAnalyst( int width, int height )
{
        int i ;
        DataAnalyst *ptr ;
        ptr = (DataAnalyst *) malloc( sizeof( DataAnalyst ) );
        ptr->Bins       = (double *) malloc( width*height*sizeof(double) );
        for( i = 0 ; i < width*height ; i++ ) ptr->Bins[i] = 0.0 ;
        ptr->width      = width ;
        ptr->height  = height ;
        ptr->BadData = 0 ;
        return(ptr) ;
}

void SetDataAnalyst( DataAnalyst *ptr, int x0, int x1, int y0, int y1 )
{
        ptr->x0 = x0 ;
        ptr->x1 = x1 ;
        ptr->y0 = y0 ;
        ptr->y1 = y1 ;
        ptr->dx = 1.0*(x1-x0)/ptr->width  ;
        ptr->dy = 1.0*(y1-y0)/ptr->height ;
}

void DAStoreData( DataAnalyst *ptr, double x, double y )
{
        int i, j ;

        i = (int) floor( (x - ptr->x0)/ptr->dx );
        j = (int) floor( (y - ptr->y0)/ptr->dy );

        if( (i < 0 ) || (i > ptr->width-1) || (j < 0 ) || ( j > ptr->height-1 ))
                ptr->BadData++ ;
        else
                ptr->Bins[i*ptr->width+j] += 1;

}
```

```
void DANormalize( DataAnalyst *ptr )
{
   int i ;
   double L1 = 0.0     ;

   for( i = 0 ; i < ptr->width*ptr->height ; i++ ) L1 += ptr->Bins[i] ;
   for( i = 0 ; i < ptr->width*ptr->height ; i++ ) ptr->Bins[i] /= L1 ;
}
```

Picard.cpp

This program sets up a Picard iteration problem using a symbolic language. The user must supply the SetGlobals, SetValues, SetVars, and SetExpression routines.

```
int main( int argc, char **argv )
{
        FILE *outf ;
        int t ;
        int level ;
        GlobalVariables globals ;

        SetGlobals(&globals) ;
        SetValues() ;
        SetVars() ;
        SetExpression() ;

        if( (outf = fopen( globals.OutputFile, "wb" )) == NULL )
        {
                printf( "Error opening: %s\n", globals.OutputFile );
                return(0) ;
        }

        WriteNoSteps( outf, globals.TimeSteps, globals.timestep );
        for( t = 0 ; t < globals.TimeSteps ; t++ )
        {
                SetVars() ;
                for( level = 0 ; level < globals.Levels ; level++ )
                {
                        //printf( "Time&Level\t%d\t%d\n", t, level );
                        EvalExpStep() ;
                        IntPolyStep() ;
                        CopyPolyStep();
                }
                EvalValues(globals.timestep) ;
                if( (t % 4) == 0 ) WritePosition( outf ) ;
                if( (t % 100) == 0 ) printf( "TimeStep = %d (done)\n", t );
        }
        fclose(outf) ;
    return(1) ;
}
```

Picard Method User Defined Routines

```
void SetGlobals(GlobalVariables *globals)
{
        globals->TimeSteps = 10000 ;
        globals->timestep  = 0.05 ;
        globals->Levels =  3 ;
        strcpy( globals->OutputFile, "c:\\jlane\\Images\\picard.txt") ;

}

void SetExpression()
{
        Example() ;
}

void SetValues()
{
        values[0] = 10 ;
        values[1] =  5 ;
        values[2] =  40 ;
}

void Example()
{
        PushPoly( &A[0], &V[1] ) ;
        PushPoly( &A[0], &V[2] ) ;
        PushOperand( &A[0], 'M' ) ;

        PushPoly( &A[1], &V[0] ) ;
        PushPoly( &A[1], &V[2] ) ;
```

```
        PushOperand( &A[1], 'M' ) ;

        PushPoly( &A[2], &V[0] ) ;
        PushPoly( &A[2], &V[1] ) ;
        PushOperand( &A[2], 'M' ) ;


}

void Cylinder()
{
        PushPoly(       &A[0],  &V[1] ) ;
        PushScalar(  &A[0], -1 ) ;
        PushOperand( &A[0], '*' );

        PushPoly(       &A[1], &V[0] ) ;

        PushPoly(       &A[2], &V[0] ) ;
        PushPoly(       &A[2], &V[1] ) ;
        PushOperand( &A[2], 'M' );
        PushScalar(  &A[2], -4 ) ;
        PushOperand( &A[2], '*' );
}

void Cylinder2()
{
        PushPoly(       &A[0],  &V[1] ) ;
        PushScalar(  &A[0], -1 ) ;
        PushOperand( &A[0], '*' );

        PushPoly(       &A[1], &V[0] ) ;

        PushPoly(       &A[2], &V[0] ) ;
        PushPoly(       &A[2], &V[1] ) ;
        PushOperand( &A[2], 'M' );
        PushPoly(       &A[2], &V[2] ) ;
        PushScalar(  &A[2], -1 ) ;
        PushOperand( &A[2], '*' );
        PushOperand( &A[2], 'A' );
}

void Rossler()
{
        double a, b, c ;
        a = 10 ;
        b = 2 ;
        c = 4 ;

        PushPoly(       &A[0], &V[1] ) ;
        PushPoly(       &A[0], &V[2] ) ;
        PushOperand( &A[0], 'A' ) ;
        PushScalar(  &A[0], -1 ) ;
        PushOperand( &A[0], '*' );

        PushPoly(       &A[1], &V[1] ) ;
        PushScalar(  &A[1], a );
        PushOperand( &A[1], '*' );
        PushPoly(       &A[1], &V[0] ) ;
        PushOperand( &A[1], 'A' );

        PushPoly(       &A[2], &V[0] ) ;
        PushScalar(  &A[2], -c ) ;
        PushOperand( &A[2], '+' );
        PushPoly(       &A[2], &V[2] );
        PushOperand( &A[2], 'M' );
        PushScalar(  &A[2], b );
        PushOperand( &A[2], '+' );
}

void SetVars()
{
```

```
        int i ;
        for( i = 0 ; i < NumberEqs ; i++ )    FreePoly( &V[i] );
        for( i = 0 ; i < NumberEqs ; i++ )    MakePoly( &V[i], values[i] );
}

void EvalExpStep()
{
        int i ;
        for( i = 0 ; i < NumberEqs ; i++ ) EvalExpress( &A[i] );
}

void IntPolyStep()
{
        int i ;
        for( i = 0 ; i < NumberEqs ; i++ ) IntPoly( &A[i].P, values[i] );
}

void CopyPolyStep()
{
        int i ;
        for( i = 0 ; i < NumberEqs ; i++ ) CopyPoly( &V[i], &A[i].P );
}

void EvalValues(double timestep)
{
        int i ;
        for( i = 0 ; i < NumberEqs ; i++ ) values[i] = EvalPoly( &V[i], timestep ) ;
}

void WritePosition( FILE *f )
{
        fwrite( values, sizeof(double), NumberEqs, f );
}

void WriteNoSteps( FILE *f, int N, double dt )
{
        fwrite( &N,  sizeof(int),      1, f );
        fwrite( &dt, sizeof(double), 1, f );
}
```

Picard Method System Routines

```
void IntPoly( Polynomial *P, double initValue )
{
        int i ;
        double *newC ;
        P->degree++ ;
        newC = (double *) malloc( sizeof(double)*(P->degree+1) );
        for( i = 0 ; i < P->degree ; i++ )
        {
                newC[i] = P->C[i] / (P->degree - i ) ;
        }
        newC[P->degree] = initValue ;
        free( P->C ) ;
        P->C = newC ;
}

void CopyPoly( Polynomial *P, Polynomial *Q )
{
        int i ;
        if( P->C != NULL )
        {
                free( P->C ) ;
                P->C = NULL ;
        }
        P->degree = Q->degree ;
        if( P->degree != -1 )
        {
```

```
                P->C = (double *) malloc( sizeof( double )*(P->degree+1) );
                for( i = 0 ; i < P->degree+1 ; i++ )
                        P->C[i] = Q->C[i] ;
        }
}

double EvalPoly( Polynomial *P, double t )
{
        int i ;
        double value ;
        value = 0 ;
        for( i = 0 ; i < P->degree+1 ; i++ )
        {
                value *= t ;
                value += P->C[i] ;
        }
        return( value );
}

void FreePoly( Polynomial *P )
{
        if( P->C != NULL ) free( P->C ) ;
        P->C = NULL ;
        P->degree = -1 ;
}

void InitPoly( Polynomial *P )
{
        P->C = NULL ;
        P->degree = -1 ;
}

void InitPolyDegree( Polynomial *P, int degree )
{
        int i ;
        P->C = (double *) malloc( sizeof(double)*(degree+1) ) ;
        P->degree = degree ;
        for( i = 0 ; i < P->degree+1 ; i++ ) P->C[i] = 0 ;
}

void MakePoly( Polynomial *P, double coeff   )
{
        if( P->C == NULL )
        {
                P->C = (double *) malloc( sizeof(double) ) ;
                P->C[0] = coeff ;
                P->degree = 0 ;
        }
        else
        {
                free( P->C ) ;
                P->C = NULL ;
                MakePoly( P, coeff );
        }
}

void PrintPoly( Polynomial *P )
{
        int i ;
        printf( "Degree: %d\t", P->degree );
        for( i = 0 ; i < P->degree+1 ; i++ ) printf( "%lg\t", P->C[i] );
        printf( "\n" );
}

void MultScalar( Polynomial *P, double scalar )
{
        int i ;
        for( i = 0 ; i < P->degree+1 ; i++ ) P->C[i] *= scalar ;
}

void AddScalar( Polynomial *P, double scalar )
```

```
{
        P->C[P->degree] += scalar ;
}

void AddPoly( Polynomial *P, Polynomial *Q )
{
        int i ;
        Polynomial A, B ;

        InitPoly( &A );
        InitPoly( &B );

        if( P->degree > Q->degree )
        {
                CopyPoly( &A, P ) ;
                CopyPoly( &B, Q ) ;
        }
        else
        {
                CopyPoly( &A, Q ) ;
                CopyPoly( &B, P ) ;
        }
        for( i = 0 ; i < B.degree+1 ; i++ )
                A.C[A.degree-B.degree+i] += B.C[i] ;

        CopyPoly( P, &A ) ;
        FreePoly( Q ) ;
        FreePoly( &A ) ;
        FreePoly( &B ) ;
}

void MultPoly( Polynomial *P, Polynomial *Q )
{
        int i, j ;
        Polynomial A ;
        int degree ;
        int newDegree ;

        newDegree = P->degree+Q->degree ;
        InitPolyDegree( &A, newDegree );
        for( i = 0 ; i < P->degree+1 ; i++ )
        for( j = 0 ; j < Q->degree+1 ; j++ )
        {
                degree = (P->degree-i) + (Q->degree-j) ;
                // this index is really just (i+j)
                A.C[newDegree-degree] += P->C[i]*Q->C[j] ;
        }

        CopyPoly( P, &A ) ;
        FreePoly( Q ) ;
        FreePoly( &A ) ;
}

void PushOperand( Expression *E, char c )
{
        Token *T ;
        T = (Token *) malloc( sizeof( Token ) );
        T->next = NULL ;
        T->type = 'O' ;
        T->Poly = NULL ;
        T->scalar = 0 ;
        T->operand = c ;
        AddToken( E, T );
}

void PushPoly( Expression *E, Polynomial *P )
{
        Token *T ;
        T = (Token *) malloc( sizeof( Token ) );
        T->next = NULL ;
        T->type = 'P' ;
```

```
        T->Poly =  P ;
        T->scalar = 0 ;
        T->operand = 0 ;
        AddToken( E, T );
}

void PushScalar( Expression *E, double scalar )
{
        Token *T ;
        T = (Token *) malloc( sizeof( Token ) );
        T->next = NULL ;
        T->type = 'S' ;
        T->Poly = NULL ;
        T->scalar = scalar ;
        T->operand = 0 ;
        AddToken( E, T );
}


void AddToken( Expression *E, Token *T )
{
        Token *S ;
        S = E->List ;
        if( S == NULL ) E->List = T ;
        else
        {
                while( S->next != NULL )
                {
                        S = S->next ;
                }
                S->next = T ;
        }
}

void EvalExpress( Expression *E )
{
        Token *T ;
        //Polynomial P, Q ;
        PolyList L ;
        L.next = NULL ;
        T = E->List ;
        while( T != NULL )
        {
                switch( T->type )
                {
                case 'O' : // operand
                        switch( T->operand )
                        {
                        case 'A' : // add poly
                                PopPolyList( &L, &E->P );
                                PopPolyList( &L, &E->Q );
                                AddPoly( &E->P, &E->Q ) ;
                                PushPolyList( &L, &E->P );
                                break ;
                        case 'M' : // mult poly
                                PopPolyList( &L, &E->P );
                                PopPolyList( &L, &E->Q );
                                MultPoly( &E->P, &E->Q ) ;
                                PushPolyList( &L, &E->P );
                                break ;
                        case '+' : // add scalar and poly
                                PopPolyList( &L, &E->P );
                                AddScalar( &E->P, E->scalar );
                                PushPolyList( &L, &E->P );
                                break ;
                        case '*' : // mult scalar and poly
                                PopPolyList( &L, &E->P );
                                MultScalar( &E->P, E->scalar );
                                PushPolyList( &L, &E->P );
                                break ;
                        default : break ;
```

```
                        )
                        break ;

                case 'S' : // scalar
                        E->scalar = T->scalar ;
                        break ;

                case 'P' : // polynomial
                        PushPolyList( &L, T->Poly );
                        break ;

                default : break ;

                }
                T = T->next ;
        }
        PopPolyList( &L, &E->P );
}

void PushPolyList( PolyList *L, Polynomial *P )
{
        PolyList *newItem ;
        newItem = (PolyList *) malloc( sizeof( PolyList ) );
        InitPoly( &newItem->P );
        if( L->next == NULL )
        {
                newItem->next = NULL ;
                CopyPoly(&newItem->P, P) ;
                L->next = newItem ;
        }
        else
        {
                newItem->next = L->next ;
                CopyPoly(&newItem->P, P) ;
                L->next = newItem ;
        }
}

void PopPolyList( PolyList *L, Polynomial *P )
{
        PolyList *ptr ;
        ptr = L->next ;
        FreePoly( P ) ;
        if( L->next != NULL )
        {
                CopyPoly( P, &L->next->P );
                L->next = L->next->next ;
                FreePoly( &ptr->P );
                free( ptr );
        }
}


void PrintExpress( Expression *E )
{
        Token *T ;

        T = E->List ;
        while( T != NULL )
        {
                switch( T->type )
                {
                case 'O' : // operand
                        printf( "Operand:\t%c\n", T->operand );
                        break ;

                case 'S' : // scalar
                        printf( "Scalar:\t%lg\n", T->scalar );
                        break ;

                case 'P' : // polynomial
```

```
        fwrite( &N,  sizeof(int),      1, f );
        fwrite( &dt, sizeof(double), 1, f );
}

void StepPicard( simulator *S )
{
        // regular simple step through
        double dx, dy, dz, dt, dt2, dt3 ;
        double x, y, z ;
        double a, b, c ;
        x = S->R[0] ;
        y = S->R[1] ;
        z = S->R[2] ;
        a = S->a ;
        b = S->b ;
        c = S->c ;
        dt = S->dt ;
        dt2 = dt*dt/2 ;
        dt3 = dt*dt*dt/3.0 ;

        dx = -(z+y)*dt - (x + a*y + b + z*(x-c))*dt2 ;
        dy = (x + a*y)*dt + (-z -y + a*x + a*a*y)*dt2 ;
        dz = (b + z*(x - c))*dt + ((x-c)*(b+z*(x-c)) - z*(z+y))*dt2 - (b + z*(x-
c))*(z+y)*dt3 ;
        S->R[0] += dx ;
        S->R[1] += dy ;
        S->R[2] += dz ;
}


void StepForward( simulator *S )
{
        // regular simple step through
        double dx, dy, dz, dt ;
        double x, y, z ;
        x = S->R[0] ;
        y = S->R[1] ;
        z = S->R[2] ;
        dt = S->dt ;

        dx = -(z+y)*dt ;
        dy = (x + S->a*y)*dt ;
        dz = (S->b + z*(x - S->c))*dt ;
        S->R[0] += dx ;
        S->R[1] += dy ;
        S->R[2] += dz ;
}

void setPosition( double *r, double x, double y, double z )
{
        r[0] = x ; r[1] = y ; r[2] = z ;
}
```

```
                        PrintPoly( T->Poly );
                        break ;

                default : break ;

                }
                T = T->next ;
        }
}
```

Rossler.cpp

This routine generates the Rossler attractor using a Picard method. It is not using the symbolic language code. Notice the complicated routine to approximate the next data point.

```
void SetSim( simulator *S )
{
        S->a = 0.15 ;
        S->b = 0.20 ;
        S->c = 10.0 ;
        S->dt = M_PI / 100.0 ;
        S->Init = 1000 ;
        S->T  = 33000 ;
        setPosition( S->R, 10.0, 0.0, 0.0 ) ;
}

void main(int argc, char **argv )
{
        int n, N ;
        simulator S1 ;
        FILE *outf ;

        outf = fopen( Outfile, "wb" ) ;
        if( outf == NULL ) printf( "Error opening file: %s\n", Outfile ) ;

        SetSim( &S1 );

        for( n = 0 ; n < S1.Init ; n++ )
        {
                StepPicard(&S1);
        }

        N = int (S1.T/S1.dt) ;
        WriteNoSteps( outf, N-S1.Init, S1.dt );
        for( n = S1.Init ; n < N ; n++ )
        {
                StepPicard(&S1);
                WritePosition(outf, &S1);
                //PrintPosition( stdout, &S1 );
        }
        fclose( outf );
}


void PrintPosition( FILE *f, simulator *S )
{
        fprintf( f, "%10.3lg\t%10.3lg\t%10.3lg\n", S->R[0], S->R[1], S->R[2] );
}

void WritePosition( FILE *f, simulator *S )
{
        fwrite( &(S->R), sizeof(double), 3, f );
}

void WriteNoSteps( FILE *f, int N, double dt )
{
```